Dijkstra Graphs

Lucila M. S. Bento[1], Davidson R. Boccardo[6,7], Raphael C. S. Machado[3], Flávio K. Miyazawa[5], Vinícius G. Pereira de Sá[2], and Jayme L. Szwarcfiter[1,2,4]

[1]COPPE-Sistemas – Universidade Federal do Rio de Janeiro
[2]Instituto de Matemática – Universidade Federal do Rio de Janeiro
[3]Instituto Nacional de Metrologia, Qualidade e Tecnologia – Inmetro
[4]Instituto de Matemática e Estatística – Universidade do Estado do Rio de Janeiro
[5]Instituto de Computação – Universidade Estadual de Campinas
[6]Clavis Segurança da Informação
[7]Green Hat Segurança da Informação
E-mails: lucilabento@ppgi.ufrj.br, davidson@clavis.com.br, rcmachado@inmetro.gov.br, fkm@ic.unicamp.br, vigusmao@dcc.ufrj.br, jayme@nce.ufrj.br

Abstract. We revisit a concept that has been central in some early stages of computer science, that of *structured programming*: a set of rules that an algorithm must follow in order to acquire a structure that is desirable in many aspects. While much has been written about structured programming, an important issue has been left unanswered: given an arbitrary, compiled program, describe an algorithm to decide whether or not it is *structured*, that is, whether it conforms to the stated principles of structured programming. We refer to the classical concept of structured programming, as described by Dijkstra. By employing a graph model and graph-theoretic techniques, we formulate an efficient algorithm for answering this question. To do so, we first introduce the class of graphs which correspond to structured programs, which we call *Dijkstra Graphs*. Our problem then becomes the recognition of such graphs, for which we present a greedy $O(n)$-time algorithm. Furthermore, we describe an isomorphism algorithm for Dijkstra graphs, whose complexity is also linear in the number of vertices of the graph. Both the recognition and isomorphism algorithms have potential important applications, such as in code similarity analysis.

Keywords: graph algorithms, graph isomorphism, reducibility, structured programming

1 Introduction

Structured programming was one of the main topics in computer science in the years around 1970. It can be viewed as a method for the development and description of algorithms and programs. Basically, it consists of a top-down formulation of the algorithm, breaking it into blocks or modules. The blocks are stepwise refined, possibly generating new, smaller blocks, until refinements no longer exist. The technique constraints the description of the modules to contain only three basic control structures: *sequence*, *selection* and *iteration.* The first of them corresponds to sequential statements of the algorithm; the second refers to comparisons leading to different outcomes; the last one corresponds to sets of actions performed repeatedly in the algorithm.

One of the early papers about structured programming was the article by Dijkstra "Go-to statement considered harmful" [8], which brought the idea that the unrestricted use of go-to statements is incompatible with well structured algorithms. That paper was soon followed by a discussion in the literature about go-to's, as in the papers by Knuth and Floyd [18], Wulf [34] and Knuth [17]. Other classical papers are those by Dahl and Hoare [9], Hoare [16] and Wirth [28], among others. Guidelines of structured programming were established in an article by Dijkstra [10]. The early development of programming languages containing blocks, such as ALGOL (Wirth [29]) and PASCAL (Naur [23]), was an important reason for structured programming's widespread adoption. This concept has been then further developed in papers by Kosaroju [20], describing the idea of reducibility among flowcharts. Moreover, [20] has introduced and characterized the class of *D-charts*, which in fact are graphs properly containing all those which originate from structured programming. Williams [32] also describes variations of different forms of structuredness, including the basic definitions by Dijkstra, as well as D-charts. The different forms of unstructuredness were described in papers by Williams [31] and McCabe [22]. The conversion of a unstructured flow diagram into a structured one has been considered by Williams and Ossher [33], and Oulsnam [24]. Formal aspects of structured programming include the papers by Böhm and Jacopini [4], Harel [12], and Kozen and Tseng [21]. A mathematical theory for modeling structuredness, designed for flow graphs, in general, has been described by Fenton, Whitty and Kaposi [11]. The actual influence of the concept of structured programming in the development of algorithms for solving various problems in different areas occurred right from the start, either explicitly, as in the papers by Henderson and Snow [15], and Knuth and Szwarcfiter [19], or implicitly as in the various graph algorithms by Tarjan, e.g. [25,26].

A natural question regarding structured programming is to recognize whether a given program is structured. To our knowledge, such a question has not been solved neither in the early stages of structured programming, nor later. That is the main purpose of the present paper. We formulate an algorithm for recognizing whether a given program is structured, according to Dijkstra's concept of structured programming. Note that the input comprises the binary code, not the source code. A well-known representation that comes in handy is that of the *control graph* (CFG) of a program, employed by the majority of reverse-engineering tools to perform data-flow analysis and optimizations. A CFG represents the intraprocedural computation of a function by depicting the existing links across its basic blocks. Each basic block represents a straight line in the program's instructions, ending (possibly) with a branch. An edge $A \to B$ (from the exit of block A to the start of block B) represents the program flowing from A to B at runtime.

We are then interested in the version of the recognition problem which takes as input a control flow graph of the program [1,5]: a directed graph representing the possible sequences of basic blocks along the execution of the program. Our problem thus becomes graph-theoretic: given a control flow graph, decide whether it has been produced by a structured program. We apply a reducibility method, whose reduction operations iteratively obtain smaller and smaller control flow graphs.

In this paper, we first define the class of graphs which correspond to structured programs, as considered by Dijkstra in [10]. Such a class has then been named as *Dijkstra graphs*. We describe a characterization that leads to a greedy $O(n)$ time recognition algorithm for a Dijkstra graph with n vertices. Among the potential direct applications of the proposed algorithm, we can mention software watermarking via control flow graph modifications [3,6].

Additionally, we formulate an isomorphism algorithm for the class of Dijkstra graphs. The method consists of defining a convenient code for a graph of the class, which consists of a string of integers. Such a code uniquely identifies the graph, and it is shown that two Dijkstra graphs are isomorphic if and only if their codes are the same. The code itself has size $O(n)$ and the time complexity of the isomorphism algorithm is also $O(n)$. In case the given graphs are isomorphic, the algorithm exhibits the isomorphism function between the graphs. Applications of isomorphism include code similarity analysis [7], since the method can determine whether apparently distinct control flow graphs (of structured programs) are actually structurally identical, with potential implications in digital rights management.

2 Preliminaries

In this paper, all graphs are finite and directed. For a graph G, we denote its vertex and edge sets by $V(G)$ and $E(G)$, respectively, with $|V(G)| = n$, $|E(G)| = m$. For $v, w \in V(G)$, an edge from v to w is written as vw. We say vw is an *out-edge* of v and an *in-edge* of w, with w an *out-neighbor* of v, and v an *in-neighbor* of w. We denote by $N^+_G(v)$ and $N^-_G(v)$ the sets of out-neighbors and in-neighbors of v, respectively. We may drop the subscript when the graph is clear from the context. Also, we write $N^{2+}(v)$ meaning $N^+(N^+(v))$. For $v, w \in V(G)$, v *reaches* w when there is a path in G from v to w. A *source* of G is a vertex that reaches all other vertices in G, while a *sink* is one which reaches no vertex, except itself. Denote by $s(G)$ and $t(G)$, respectively, a source and a sink of G. A *(control) flow* graph G is one which contains a distinguished source $s(G)$. A *source-sink* graph contains both a distinguished source $s(G)$ and distinguished sink $t(G)$. A *trivial* graph contains a single vertex.

A graph with no directed cycles is called *acyclic*. In an acyclic graph if there is a path from vertex v to vertex w, then v is an ancestor of w, and the latter a descendant of v. Additionally, if v, w are distinct then v is a it proper ancestor, and w a *proper descendant*. Let G be a flow graph with source $s(G)$, and C a cycle of G. The cycle C is called a *single-entry cycle* if it contains a vertex

$v \in C$ that separates $s(G)$ from the vertices of $C \setminus \{v\}$. A flow graph in which each of its cycles is a single-entry cycle is called *reducible.* Reducible graphs were characterized by Hecht and Ullman [13,14]. An efficient recognition algorithm for this class has been described by Tarjan [27].

In a *depth-first search (DFS)* of a directed graph, in each step a vertex is inserted in a stack, or removed from it. Every vertex is inserted and removed from the stack exactly once. An edge $vw \in E(G)$, such that v is inserted in the stack after w, and before the removal of w, is called a *cycle edge.* Let C be the set of cycle edges of a graph, relative to some DFS. Clearly, the graph $G - C$ is acyclic. The following characterization if reducible flow graphs is relevant for our purposes.

Theorem 1 *[14,27] A flow graph G is reducible if and only if, for any depth-first search of G starting from $s(G)$, the set of cycle edges is invariant.*

In a flow graph graph G, we may write DFS of G, as to mean a DFS of G staring from $s(G)$. In addition, if G is also reducible, based of the above theorem, we may use the terms *ancestor* or descendant of G, as to mean *ancestor* or descendant of $G - C$, where C is the (unique) set of cycle edges of G.

A *topological sort* of a graph G is a sequence $v_1, \ldots, v_n$ of its vertices, such that $v_i v_j \in E(G)$ implies $i < j$. It is well known that G admits a topological sort if and only if G is acyclic. Finally, two graphs G_1, G_2 are *isomorphic* when there is a one-to-one correspondence $f : V(G_1) \cong V(G_2)$ such that $vw \in E(G_1)$ if and only if $f(v)f(w) \in E(G_2)$. In this case, write $G_1 \cong G_2$, and call f an *isomorphism function* between G_1, G_2, with $f(v)$ being the *image* of v under f.

3 The Graphs of Structured Programming

In this section, we describe the graphs of structured programming, as established by Dijkstra in [10], leading to the definition of *Dijkstra graphs.* First, we introduce a family of graphs directly related to Dijkstra's concepts of structured programming.

A *statement graph* is defined as being one of the following:

(a) *trivial* graph
(b) *sequence* graph
(c) *if* graph
(d) *if-then-else* graph
(e) *p-case* graph, $p \geq 3$
(f) *while* graph
(g) *repeat* graph

For our purposes, it is convenient to assign labels to the vertices of statement graphs as follows. Each vertex is either an *expansible vertex*, labeled X, or a *regular vertex*, labelled R. See Figures 1 and 2, where the statement graphs are depicted with the corresponding vertex labels. All statement graphs are source-sink. Vertex v denotes the source of the graph in each case.

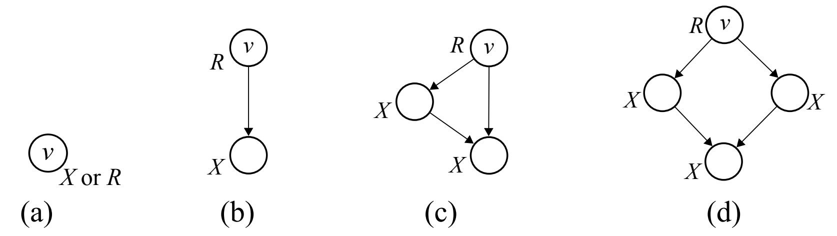


Fig. 1: Statement graphs (a)-(d)

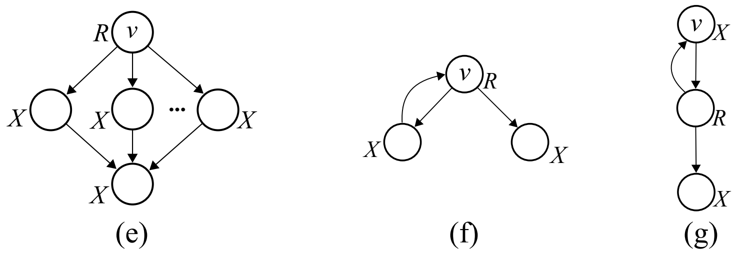


Fig. 2: Statement graphs (e)-(g)

Let G be an unlabeled reducible graph, and H a subgraph of G, having source $s(H)$ and sink $t(H)$. We say H is *closed* when

- $v \in V(H) \setminus s(H) \Rightarrow N^-(v) \subseteq V(H)$;
- $v \in V(H) \setminus t(H) \Rightarrow N^+(v) \subseteq V(H)$; and
- $vs(H)$ is a cycle edge $\Rightarrow v \in N^+(s(H))$.

In this case, $s(H)$ is the only vertex of H having possible in-neighbors outside H, and $t(H)$ the only one possibly having out-neighbors outside H.

The following concepts are central to our purposes.

Let H be an induced subgraph of G. We say H is *prime* when

- H is isomorphic to some non-trivial statement graph, and
- H is closed.

It should be noted that the while and repeat graphs, respectively, (f) and (g) of Figure 2, are not isomorphic in the context of flow reducible graphs. In fact, the cycle edge turns them distinguishable. The sources of such graphs are the entry vertices of the cycle edge, respectively. Then the sink is an out-neighbor of the source in (f), but not in (g).

Next, let G, H be two graphs, $V(G) \cap V(H) = \emptyset$, H source-sink, $v \in V(G)$.

The *expansion* of v into a source-sink graph H (Figure 3) consists of replacing v by H, in G, such that

- $N_G^-(s(H)) := N_G^-(v)$;
- $N_G^+(t(H)) := N_G^+(v)$; and

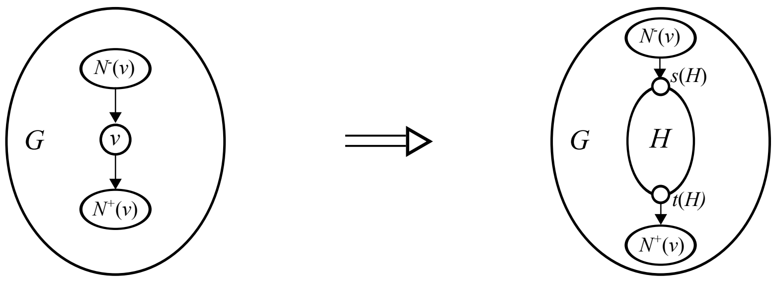


Fig. 3: Expansion operation

– the remaining adjacencies are unchanged.

Now let G be a graph, and H a prime subgraph of G. The *contraction* of H into a single vertex (Figure 4) is the operation defined by the following steps:

1. Identify (coalesce) the vertices of H into the source $s(H)$ of H.
2. Remove all parallel edges and loops.

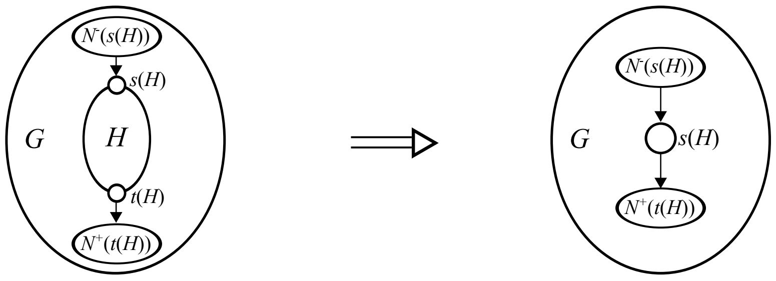


Fig. 4: Contraction operation

We finally have the elements to define the class of Dijkstra graphs. The concepts of structured programming and top-down refinement [10] lead naturally to the following definition.

A *Dijkstra graph (DG)* has vertices labeled X or R recursively defined as:

1. A trivial statement graph is a DG.
2. Any graph obtained from a DG by expanding some X-vertex into a non-trivial statement graph is also a DG. Furthermore, after expanding an X-labeled vertex v into a statement graph H, vertex $s(H)$ is labeled as R.

An example is given in Figure 5.

The above definition leads directly to a method for constructing Dijkstra graphs, as follows. Find a sequence of graphs $G_0, \ldots, G_k$, such that

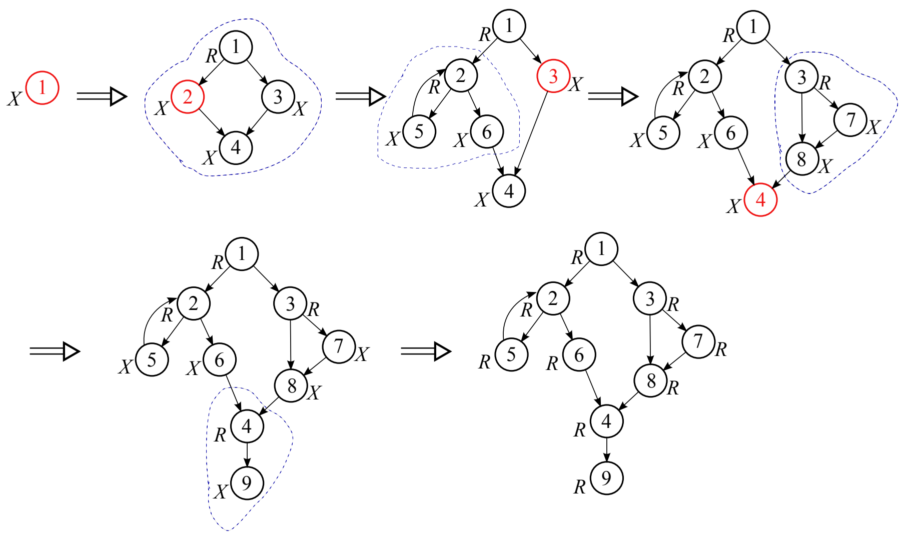


Fig. 5: Obtaining a Dijkstra graph via vertex expansions

– G_0 is the trivial graph, with the vertex labeled X;
– G_i is obtained from G_{i-1}, $i \geq 1$, by expanding some X-vertex v of it into a statement graph H.

The above construction does not imply a polynomial-time algorithm for recognizing graphs of the class. In the next section, we describe another characterization which leads to such an algorithm. It is relevant to emphasize that the labels are used merely for constructing the graphs. For the actual recognition process, we are interested in the problem of deciding whether a given *unlabeled* flow graph is actually a Dijkstra graph.

4 Recognition of Dijkstra Graphs

In this section, we describe an algorithm for recognizing Dijkstra graphs. For the recognition process, the hypothesis is that we are given an arbitrary flow graph G, with no labels, and the aim is to decide whether or not G is a DG. First, we introduce some notation and describe the propositions which form the basis of the algorithm.

4.1 Basic Lemmas

The following lemma states some basic properties of Dijkstra graphs.

Lemma 2 *If G is a Dijkstra graph, then*

(i) G contains some prime subgraph;

(ii) G is a source-sink graph; and
(iii) G is reducible.

Proof. By definition, there is a sequence of graphs $G_0, \ldots, G_k$, where G_0 is trivial, $G_k = G$ and G_i is obtained from G_{i-1} by expanding some X-vertex $v_{i-1} \in V(G_{i-1})$ into a statement graph $H_i \subseteq G_i$. Then no vertex $v_i \in V(H_i)$, except $s(H_i)$ has in-neighbors outside H_i, and also no vertex $v_i \in V(H_i)$, except $t(H_i)$, has out-neighbors outside H_i. Furthermore, if H_i contains any cycle then H_i is necessarily a while graph or a repeat graph. The latter implies that such a cycle is $s(H)v$, where $v \in N^+(s(H))$. Therefore H_i is prime in G_i meaning that (i) holds. To show (ii) and (iii), first observe that any statement graph is single-source and reducible. Next, apply induction. For G_0, there is nothing to prove. Assume it holds for G_i, $i > 1$. Let $v_{i-1} \in V(G_{i-1})$ be the vertex that expanded into the subgraph $H_i \subseteq G_i$. Then the external neighborhoods of H_i coincide with the neighborhoods of v_{i-1}, respectively. Consequently, G_i is single-source. Now, let C_i be any cycle of G_i, if existing. If $C_i \cap H_i = \emptyset$ then C_i is single-entry, since G_{i-1} is reducible. Otherwise, if $C_i \subset V(H_i)$ the same is valid, since any statement graph is reducible. Finally, if $C_i \not\subset V(H_i)$, then v_{i-1} is contained in a single-entry cycle C_{i-1} of G_{i-1}. Then C_i has been formed by C_{i-1}, replacing v_{i-1} by a path contained in H_i. Since C_{i-1} is single-entry, it follows that C_i must be so.

Denote by $\mathcal{H}(G)$ the set of non-trivial prime graphs of G. Let $H, H' \in \mathcal{H}(G)$. Call H, H' *independent* when

- $V(H) \cap V(H') = \emptyset$, or
- $V(H) \cap V(H') = \{v\}$, where $v = s(H) = t(H')$ or $v = t(H) = s(H')$.

The following lemma assures that any pair of distinct, non-trivial prime subgraphs of a graph consists of independent subgraphs.

Lemma 3 *Let $H, H' \in \mathcal{H}$. It holds that H, H' are independent.*

Proof. If $V(H) \cap V(H') = \emptyset$ the lemma holds. Otherwise, let $v \in V(H) \cap V(H')$. The alternatives $v = s(H_1) = s(H_2)$, $v = t(H_1) = t(H_2)$, $v \neq s(H_1), t(H_1)$ or $v \neq s(H_2), t(H_2)$ do not occur because they imply H_1 or H_2 not to be closed. Next, let $v_1, v_2 \in V(H_1) \cap V(H_2)$, $v_1 \neq v_2$. In this situation, examine the alternative where $v_1 = s(H_1) = t(H_2)$ and $v = s(H_2) = t(H_1)$. The latter implies that exactly one of H_1 or H_2, say H_2, is a while graph or a repeat graph. Then there is a cycle edge $ws(H_1)$, satisfying $w \in N^-(s(H_1))$ and $w \in V(H_2) \backslash \{t(H_2)\}$. Consequently, $w \notin N^+(s(H_1))$, contradicting H_1 to be closed. The only remaining alternative is $V(H_1) \cap V(H_2) = \{v\}$, with $v = s(H_1) = t(H_2)$ or $v = s(H_2) = t(H_1)$. Then H_1, H_2 are indeed independent (see Figure 6).

Next, we introduce a concepts which central for the characterization.

Let G be a graph, $\mathcal{H}(G)$ the set of non-trivial prime subgraphs of G, and $H \in \mathcal{H}(G)$. Denote by $G \downarrow H$ the graph obtained from G by contracting H. For $v \in V(G)$, the *image* of v in $G \downarrow H$, denoted $I_{G\downarrow H}(v)$, is

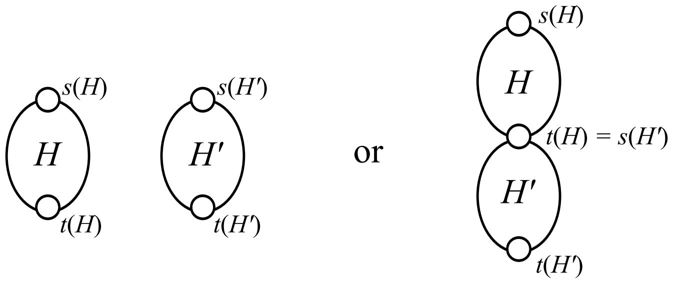


Fig. 6: Independent primes

$$I_{G\downarrow H}(v) = \begin{cases} v, & \text{if } v \notin V(H) \\ s(H), & \text{otherwise.} \end{cases}$$

For $V' \subseteq V(G)$, define the (*subset*) *image* of V' in $G \downarrow H$, as $I_{G\downarrow H}(V') = \cup_{v\in V'} I_{G\downarrow H}(v)$. Similarly, for $H' \subseteq G$, the (*subgraph*) *image* of H' in $G \downarrow H$, denoted by $I_{G\downarrow H}(H')$, is the subgraph induced in $G \downarrow H$ by the subset of vertices $I_{G\downarrow H}(V(H'))$.

The following lemmas are employed in the ensuing characterization. The first shows that any prime subgraph $H \in \mathcal{G}$ is preserved under contractions of different primes. Let G be an arbitrary flow graph, $H, H' \in \mathcal{H}(G)$, $H \neq H'$.

Lemma 4 *$I_{G\downarrow H}(H') \in \mathcal{H}(G \downarrow H)$.*

Proof. Let G be a graph, $H, H' \in \mathcal{H}(G)$, $H \neq H'$. By Lemma 3, H, H' are independent. If H, H' are disjoint the contraction of H does not affect H', and the lemma holds. Otherwise, by the independence condition, it follows that $V(H) \cap V(H') = \{v\}$, where $v = s(H) = t(H')$ or $v = s(H') = t(H)$. Examine the first of these alternatives. By contracting H, all neighborhoods of the vertices of $I_{G\downarrow H}(H')$ remain unchanged, except that of $I_{G\downarrow H}(s(H'))$, since its in-neighborhood becomes equal to $N^-_G(s(H))$. On the other hand, the contraction of H into v cannot introduce new cycles in H'. Consequently, H' preserves in $G \downarrow H$ its property of being a non-trivial and closed statement graph, moreover, prime. Finally, suppose $v = s(H) = t(H')$. Again, the neighborhoods of the vertices of $I_{G\downarrow H}(H')$ are preserved, except possibly the out-neighborhoods of the vertices of $I_{G\downarrow H}(t(H'))$, which become $N^+_G(t(H))$, after possibly removing self-loops. Consequently, $I_{G\downarrow H}(H') \in \mathcal{H}(G \downarrow H)$.

Next we prove prove a commutative law for the order of contractions.

Lemma 5 *If $H, H' \in \mathcal{H}(G)$, then $(G \downarrow H) \downarrow (I_{G\downarrow H}(H')) \cong (G \downarrow H') \downarrow (I_{G\downarrow H'}(H))$.*

Proof. Let $A \cong (G \downarrow H) \downarrow (I_{G\downarrow H}(H'))$ and $B \cong (G \downarrow H') \downarrow (I_{G\downarrow H'}(H))$. By Lemma 3, H, H' are independent. First, suppose H, H' are disjoint. Then

$I_{G\downarrow H}(H') = H'$ and $I_{G\downarrow H'}(H) = H$. It follows that, in both graphs A and B, the subgraphs H and H' are respectively replaced by a pair of non-adjacent vertices, whose in-neighborhoods are $N^-_G(s(H))$ and $N^-_G(s(H'))$, and out-neighborhoods $N^+_G(t(H))$ and $N^+_G(t(H'))$, respectively. Then $A = B$. In the second alternatives, suppose H, H' are not disjoint. Then $V(H) \cap V(H') = \{v\}$, where $v = s(H) = t(H')$, or $v = t(H) = s(H')$. In both cases, and in both graphs A and B, the subgraphs H and H' are contracted into a common vertex w. When $v = s(H) = t(H')$, it follows $N^-_G(A) = N^-_G(s(H')) = N^-_B(v)$ and $N^+_A(v) = N^+_G(t(H)) = N^+_B(v)$. Finally, when $v = t(H) = s(H')$, we have $N^-_A(v) = N^-_G(s(H)) = N^-_B(v)$, while $N^+_A(v) = N^+_G(t(H')) = N^+_B(v)$. Consequently, $A = B$ in any situation. □

4.2 Contractile Sequences

A sequence of graphs $G_0, \dots, G_k$ is a *contractile sequence* for a graph G, when

- $G \cong G_0$, and
- $G_{i+1} \cong (G_i \downarrow H_i)$, for some $H_i \in \mathcal{H}(G_i)$, $i < k$. Call H_i the *contracting prime* of G_i.

We say $G_0, \dots, G_k$ is *maximal* when $\mathcal{H}(G_k) = \emptyset$. In particular, if G_k is the trivial graph then $G_0, \dots, G_k$ is maximal.

Let $G_0, \dots, G_k$, be a contractile sequence of G, and H_j the contracting prime of G_j. That is, $G_{j+1} \cong (G_j \downarrow H_j)$, $0 \leq j < k$. For $H'_j \subseteq G_j$ and $q \geq j$, the *iterated image* of H'_j in G_q is recursively defined as

$$I_{G_q}(H'_j) = \begin{cases} H'_j, & \text{if } q = j \\ I_{G_q}(I_{G_{j+1}}(H'_j)), & \text{otherwise.} \end{cases}$$

Finally, we describe the characterization in which the recognition algorithm for Dijkstra graphs is based.

Theorem 6 *Let G be an arbitrary flow graph, with $G_0, \dots, G_k$ and $G'_0, \dots, G'_{k'}$ two contractile sequences of G. Then $G_k \cong G'_{k'}$. Furthermore, $k = k'$.*

Proof. Let $G_0, \dots, G_k$ and $G'_0, \dots, G'_{k'}$ be two contractile sequences, denoted respectively by S and S' of a graph G. Let H_j and H'_j be the contracting primes of G_j and G'_j, respectively. That is, $G_{j+1} \cong (G_j \downarrow H_j)$ and $G'_{j+1} \cong (G'_j \downarrow H'_j)$, $j < k$ and $j < k'$. Without loss of generality, assume $k \leq k'$. Let i be the least index, such that $G_j \cong G'_j$, $j \leq i$. Such an index exists since $G \cong G_0 \cong G'_0$. If $i = k$ then $G_k \cong G'_{k'}$, implying $k = k'$ and the theorem holds. Otherwise, $i < k$, $G_i \cong G'_i$ and $G_i \ncong G'_i$. Since $G_i \cong G'_i$, it follows $H_i \in \mathcal{H}(G'_i)$. By Lemma 4, the iterated image H_{i_q}, of H_i in G'_q is preserved as a prime subgraph for all G'_q, as long as it does not become the contracting prime of G'_{q-1}. Since $G'_{k'}$ has no prime subgraph, it follows there exists some index p, $i < p < k'$, such that $G'_{p+1} \cong (G_p \downarrow H_{i_p})$, where H_{i_p} represents the iterated image of H_i in G'_p. Let $H_{i_{p-1}}$ be the iterated image of H_i in G'_{p-1}. Clearly, $H'_{p-1}, H_{i_{p-1}} \in \mathcal{H}(G'_{p-1})$, and by Lemma 3, H'_{p-1} and $H_{i_{p-1}}$ are independent in G'_{p-1}. Since $((G'_{p-1} \downarrow H'_{p-1}) \downarrow H_{i_p}) \cong G'_{p+1}$, by Lemma 4, it follows that $((G'_{p-1} \downarrow H_{i_{p-1}}) \downarrow H''_{p-1}) \cong G'_{p+1}$, where H''_{p-1} represents the image of H'_{p-1} in $G'_{p-1} \downarrow H_{i_{p-1}}$. Consequently, we have exchanged the positions in S' of two contracting primes, respectively at indices $p-1$ and p, while preserving all graphs G'_q, for $q < p-1$ and $q > p$. In particular, preserving the graph G'_{p+1} and all graphs lying after G'_{p+1} in S', together with their corresponding contracting primes.

Finally, apply the above operation iteratively, until eventually the iterated image of H_i becomes the contracting prime of G'_i. In the latter situation, the two sequences coincide up to index $i+1$, while preserving the original graphs G_k and $G'_{k'}$. Again, applying iteratively such an argument, we eventually obtain that the two sequences turned coincident, preserving the original graphs G_k and $G'_{k'}$. Consequently, $G_k \cong G'_{k'}$ and $k = k'$.

4.3 The Recognition Algorithm

We start with a bound for the number m of edges of Dijkstra graphs.

Lemma 7 *Let G be a DG graph. Then $m \leq 2n-2$.*

Proof. : If G is a DG graph there is a sequence of graphs $G_0, \dots G_k$, where G_0 is the trivial graph, $G_k \cong G$ and G_i is obtained from G_{i-1} by expanding an X-vertex of G_{i-1} into a statement graph. Apply induction on the number of expansions employed in the construction of G. If $k = 0$ then G is a trivial graph, which satisfies the lemma. For $k \geq 0$, Suppose the lemma true for any graph $G' \cong G_i$, $i < k$. In particular, let $G_i \cong G_{k-1}$. Let n' and m' be the number of vertices and edges of G', respectively. Then $m' \leq 2n' - 2$. We know that G_k has been obtained by expanding a vertex of G_{k-1} into a statement graph H. Discuss the alternatives for H. If H is the trivial graph then $n = n'$ and $m = m'$. If H is a sequence graph then $n = n' + 1$ and $m = m' + 1$. If H is an if graph, a while graph or repeat graph then $n = n' + 2$ and $m = m' + 3$. If H is an if then else graph or a p-case graph then $n = n' + p + 1$ and $m = m' + 2p$, where p is the outdegree of the source of H. In any of these alternatives, a simple calculation implies $m \leq 2n - 2$.

We can describe an algorithm for recognizing Dijkstra graphs based on Theorem 6. We recall that the input is a unlabeled flow graph with no labels. Furthermore, for a while, assume that G is reducible, otherwise by Lemma 2 it is surely not a Dijkstra graph.

Let G be a flow reducible graph. To apply Theorem 6, we construct a contractile sequence $G_0, \dots, G_k$ of G. That is, find iteratively a non-trivial prime subgraph H_i of the G_i and contract it, until either the graph becomes trivial or otherwise no such subgraph exists. In the first case the graph is a DG, while in the second it is not. Recall from Lemma 4 that whenever G_i contains another prime $H_j \neq H_i$ then the iterated image of H_j is preserved, as long as it does not become the contracting prime in some later iteration. On the other hand, the contraction $G_i \downarrow H_i$ may generate a new prime H'_i, as shown in Figure **??**.

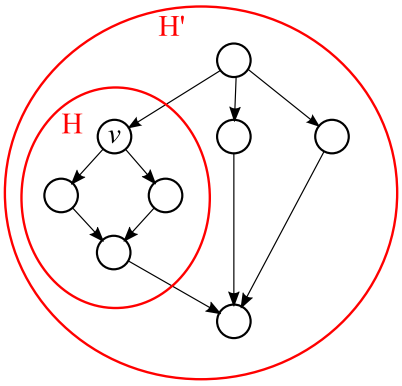


Fig. 7: Generating a new prime H'

However, the generation of new primes obeys a rule, described by the lemma below.

Lemma 8 *Let G be reducible graph, $H \in \mathcal{H}(G)$, $H' \in \mathcal{H}(G \downarrow H) \setminus \mathcal{H}(G)$. Then $s(H)$ is a proper descendant of $s(H')$ in $G \downarrow H$.*

The above lemma suggests us to consider special contractile sequences, as below.

Let G be a reducible graph, $G_0, \ldots, G_k$ a contractile sequence $\mathcal{C}$ of G, H_i the contracting prime of G_i, $0 \leq i < k$. Say that $\mathcal{C}$ is a *bottom-up (contractile) sequence* of G when each contracting prime H_i satisfies: $s(H_i)$ is not a descendant of $s(H)$, for any prime $H \neq H_i$ of G_i.

The idea of the recognition algorithm then becomes as follows. Let G be a reducible graph. Iteratively, find a lowest vertex v of G, s.t. v is the source of a prime subgraph H of G. Then contract H. Stop when noprimes exist any more.

A complete description of the algorithm is below detailed. The algorithm answers YES or NO, according to respectively G is a Dijkstra graph or not.

The correctness of Algorithm 1 follows basically from Theorem 6 and Lemma 8. However, the latter relies on the fact that G is a reducible graph, whereas the proposed algorithm considers as input an arbitrary graph. The lemma below justifies that can we avoid the step of recognizing reducible graphs.

Lemma 9 *Let G be an arbitrary flow graph input to Algorithm 1. If G is not a reducible graph then the algorithm would correctly answer NO.*

Proof. If G is not a reducible graph let E_C be the set of cycle edges, relative to some DFS startingate $s(G)$. Then G contains some cycle C, such that w does not separate $s(G)$ from v, where $vw \in E_C$ is the cycle edge of C. Without loss of generality, consider the inner most of these cycles. The only way in which the edge vw, or any of its possible images, can be contracted is in context the of a while or repeat prime subgraph H, in which the cycle would be contracted into

Algorithm 1: Dijkstra graphs recognition algorithm

```
$G$, arbitrary flow graph (no labels)
Count the number $m$ of edges of $G$. If $m \geq 2n-1$ then **return** NO
$E_C$, set of cycle edges of a DFS of $G$, starting at $s(G)$
$v_1, \ldots, v_n$, topological sorting of $G - E_C$
$i := n$
**while** $i \geq 1$ **do**
    **if** $G$ is the trivial graph
        **then return** YES, **stop**
    if $v_i$ is the source of a prime subgraph $H$ of $G$
        **then** $G := G \downarrow H$
    $i := i - 1$
**return** NO
```

vertex w, or a possible iterated image of it. However there is no possibility for H to be identified as such, because the edge entering the cycle from outside prevents the subgraph to be closed. Consequently, the algorithm necessarily would answer NO.

As for the complexity, first observe that to decide whether the graph contains a non-trivial prime subgraph whose source is a given vertex $v \in V(G)$, we need $O|(N^+(v)|$ steps. Therefore, when considering all vertices of G we require $O(m)$ time. There can be $O(n)$ prime subgraphs altogether, and each time some prime H is identified, it is contracted, and the size of the graph decreases by $|E(H)|$. The number of steps required to contract a H is $O|E(H)|$. Hence each edge is examined at most a constant number of times during the entire process. Finding a topological sorting of a graph can be done in $O(m)$. Thus, the time complexity is $O(m)$, that is, $O(n)$, by Lemma 7.

5 Isomorphism of Dijkstra Graphs

In this section, we describe a linear time algorithm for the isomorphism of Dijkstra graphs.

Given a Dijkstra graph G, the general idea consists of defining a code $C(G)$ for G, having the following property. For any two Dijkstra graphs G_1, G_2, $G_1 \cong G_2$ if and only if $C(G_1) = C(G_2)$.

As in the recognition algorithm, the codes are obtained by constructing a bottom-up contractile sequence of each graph. The codes refer explicitly to the statement graphs having source v as depicted in Figures 1 and 2, and consist of (linear) strings. For a Dijkstra graph G, the string $C(G)$ that will be coding G is constructed over an alphabet of symbols containing integers in the range $\{1, \ldots, \Delta^+(G) + 4\}$, where $\Delta^+(G)$ is the maximum cardinality among the out-neighborhoods of G. Let, A, B be a pair of strings. The concatenation of A and B, denoted $A||B$, is the string formed by A, immediately followed by B.

In order to define the code $C(G)$ for a Dijkstra graph G, we assign an integer, named $type(H)$, for each statement graph H, a code $C(v)$ for each vertex $v \in V(G)$, and a code $C(H)$ for each prime subgraph H of a bottom-up contractile sequence of G. The code $C(G)$ of the graph G is defined as being that of the source of G. For a subset $V' \subseteq V(G)$, the code $C(V')$ of V' is the set of strings $C(V') = \{C(v_i)|v_i \in V'\}$. Write $lex(C(V')) = C(v_1)||...||C(v_r)$ whenever $V' = \{v_1, \ldots, v_r\}$ and $C(v_i)$ is lexicographically not greater than $C(v_{i+1})$.

Table 1: Statement graph types and codes $C(H)$ of prime subgraphs H

statement graphs H	$type(H)$	$C(H), v = s(H)$
trivial	1	
sequence	2	$2\|\|C(N^+(v))$
if-then	3	$3\|\|C(N^+(v)) \setminus N^{+2}(v))\|\|C(N^{+2}(v))$
while	4	$4\|\|C(N^+(v) \cap N^-(v))\|\|C(N^+(v) \setminus N^-(v))$
repeat	5	$5\|\|C(N^+(v))\|\|C(N^{+2}(v) \setminus \{v\})$
if-then-else	6	$6\|\|lex(C(N^+(v)))\|\|C(N^{+2}(v))$
p-case	$p + 4$	$p + 4\|\|lex(C(N^+(v)))\|\|C(N^{+2}(v))$

Next, we describe how to obtain the actual codes. The types of the the different statement graphs are shown in the second column of Table 1. For a vertex $v \in V(G)$, the code $C(v)$ is initially set to 1. Subsequently, if v becomes the source of a prime graph H, the string $C(v)$ is updated by implicitly assigning $C(v) := C(v)||C(H)$, where $C(H)$ is given by the third column of the table. Such an operation is called the *expansion* of v. It follows that $C(H)$ is written in terms of $type(H)$ and the codes of the vertices of H, and so on iteratively. A possible expansion of some other vertex $w \in V(G)$ could imply in an expansion of v, and so iteratively. Observe that when H is an if-then-else or a p-case graph, we have chosen to place the codes of the out-neighbors of $s(H)$ in lexicographic ordering. For the remaining statement graphs H, the ordering of the codes of the out-neighbors of $s(H)$ is also unique and implicitly imposed by H. When all primes associated to $C(v)$ have been expanded, $C(v)$ has reached its final value,

5.1 The Isomorphism Algorithm

Next, we describe the actual formulation of the algortithm.

Let G be a DG. Algorithm 2 constructs the encoding $C(G)$ for G.

An example is given in Figure 8.

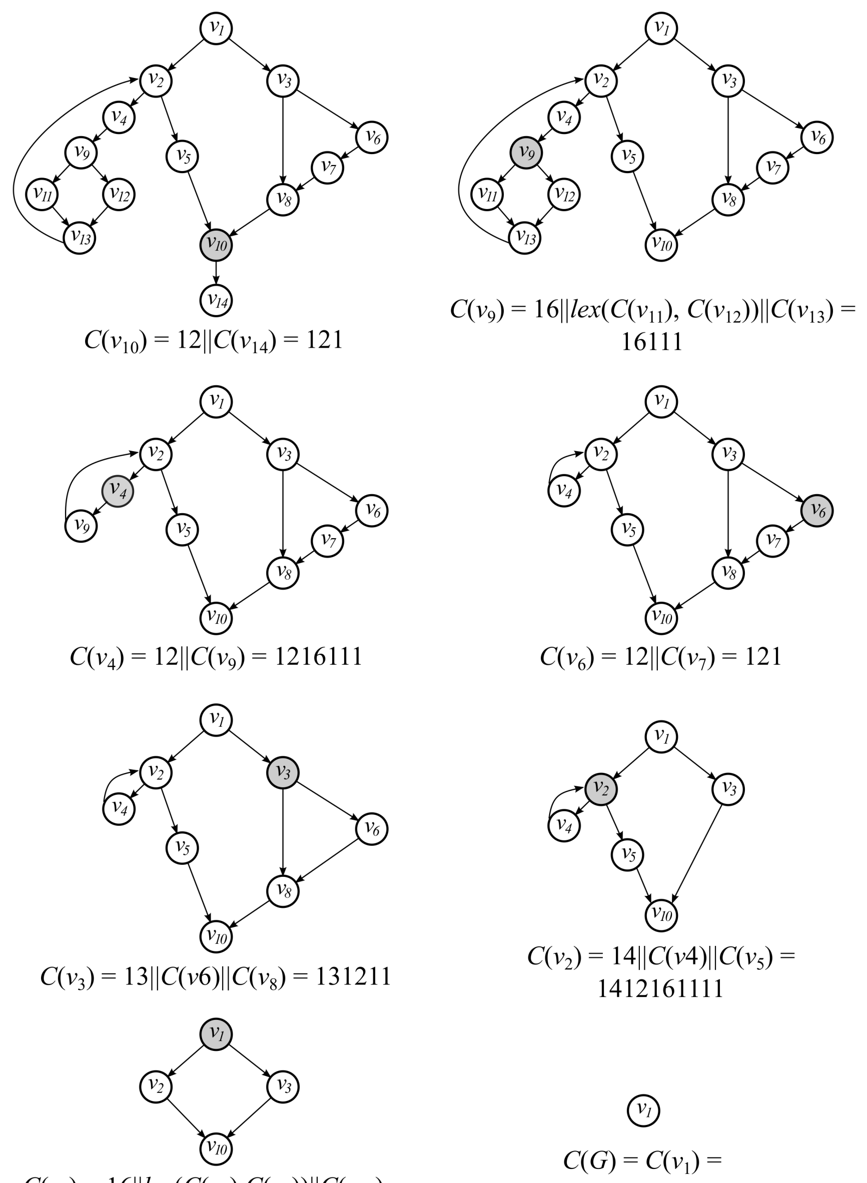
5.2 Correctness and Complexity

Theorem 10 *Let G, G' de Dijkstra graphs, and $C(G), C(G')$ their codes, respectively. Then G, G' are isomorphic if and only if $C(G) = C(G')$.*

Algorithm 2: Dijkstra graphs isomorphism algorithm

```
$G$, DG; $E_C$, set of cycle edges of $G$
Find a topological sorting $v_1, \ldots, v_n$ of $G - E_C$
for $i = n, n-1, \ldots, 1$ do
    $C(v_i) := 1$
    if $v_i$ is the source of a prime subgraph $H$ then
        $C(v_i) := C(v_i)|| \begin{cases} 2||C(N^+(v_i)), \text{ if } H \text{ is a sequence graph;} \\ 3||C(N^+(v_i) \setminus N^{+2}(v_i))||C(N^{+2}(v_i)), \\ \qquad \text{if } H \text{ is an if-then graph;} \\ 4||C(N^+(v_i) \cap N^-(v_i))||C(N^+(v_i) \setminus N^-(v_i)), \\ \qquad \text{if } H \text{ is a while graph,} \\ 5||C(N^+(v_i))||C(N^{+2}(v_i) \setminus \{v_i\}), \\ \qquad \text{if } H \text{ is a repeat graph;} \\ 6||lex(C(N^+(v_i)))||C(N^{+2}(v_i)), \\ \qquad \text{if } H \text{ is an if-then-else graph.} \\ p+4||lex(C(N^+(v_i)))||C(N^{+2}(v_i)), \\ \qquad \text{if } H \text{ is a p-case graph.} \end{cases}$
$C(G) := C(v_1)$
```

Proof. By hypothesis, G, G' are isomorphic. We show that it implies $C(G) = C(G')$. Following the isomorphism algorithm, observe that the number of 1's in the strings $C(G), C(G')$ represents the number of vertices of G, G', respectively, whereas each integer > 1 in the strings, represents the contraction of a prime subgraph. Furthermore, each prime subgraph H, which is initially contained in the input graph G, corresponds in $C(G)$, to a substring formed by the integer $type(H)$ followed by one 1, if $type(H) = 2$; or two 1's, if $type(H) = 3$; or three 1's, if $4 \leq type(H) \leq 6$; or $type(H) + 1$ 1's, if $type(H) > 6$; respectively. Clearly, the same holds for the graph G' and its code $C(G')$. The proof is by induction on the number k of contractions needed to reduce both G and G' to a trivial vertex. By Theorem 6, k is invariant and applies for both graphs G and G'. If $k = 0$ then both G and G' are trivial graphs, and the theorem holds, since $C(G) = C(G') = 1$. When $k > 0$, assume that if G_- and G'_- are isomorphic DG graphs which require less than k contractions for reduction then $C(G_-) = C(G'_-)$. Furthermore, assume also by the induction hypothesis, that if v, v' are vertices of G_-, G'_-, corresponding to 1's at the same relative positions in $C(G)$ and $C(G_-)$, respectively, then $v' = f(v)$, where f is the isomorphism function between G_- and G'_-. Now, consider the graphs G and G'. Choose a prime subgraph H of G, and let $v = s(H)$. Let $v' = f(v)$ be a vertex of G' corresponding to v by the isomorphism. Since $G \cong G'$, it follows that v' is the source of a prime subgraph H' of G'. Moreover $H \cong H'$. Consider the contractions $G \downarrow H$ and $G' \downarrow H'$, leading to graphs G_- and G'_-, respectively. Let $C_-(G)$ and $C_-(G')$ be the strings obtained from $C(G)$ and $C(G')$, respectively by contracting the substrings corresponding to H and H', as above. That is, all the 1's of $C(H)$ and $C(H')$ are compressed into the positions of $v = s(H)$ and $v' = s(H')$, respectively, while the integers $type(H)$ and $type(H')$ become 1, maitaining their



$C(v_{10}) = 12\|C(v_{14}) = 121$

$C(v_9) = 16\|lex(C(v_{11}), C(v_{12}))\|C(v_{13}) = 16111$

$C(v_4) = 12\|C(v_9) = 1216111$

$C(v_6) = 12\|C(v_7) = 121$

$C(v_3) = 13\|C(v6)\|C(v_8) = 131211$

$C(v_2) = 14\|C(v4)\|C(v_5) = 1412161111$

$C(v_1) = 16\|lex(C(v_2),C(v_3))\|C(v_{10}) = 161312111412161111121$

$C(G) = C(v_1) = 161312111412161111121$

Fig. 8: Example for isomorphism algorithm

original positions. It follows that $C(G_-) = C_-(G)$ and $C(G'_-) = C_-(G')$. By the induction hypothesis $C(G_-) = C(G'_-)$ and the 1's corresponding to v and v' lie in the same relative positions in the strings. Consequently, by replacing the latter

1's for the substrings which originally represented H and H', we conclude that indeed $C(G) = C(G')$, and moreover the induction hypothesis is still verified. The converse is similar.

The corollaries below are direct consequences of Theorem 10.

Corollary 11 *Let G be a DG. The following affirmatives hold.*

1. *There is a one-to-one correspondence between the 1's of $C(G)$ and the vertices of G.*
2. *The code $C(G)$ of G is unique and is a representation of G.*

Corollary 12 *Let G, G' be DGs and $C(G), C(G')$ their corresponding codes, satisfying $C(G) = C(G')$. Then an isomorphism function f between G and G' can be determined as follows. Let $v \in V(G)$ and $v' \in V(G')$ correspond to 1's at identical relative positions in $C(G)$ and $C(G')$, respectively. Define $f(v) := v'$.*

Finally, consider the complexity of the isomorphism algorithm.

Lemma 13 *Let G be a Dijkstra graph, and $C(G)$ its code. Then $|C(G)| = n + k \leq 2n - 1$, where n is the number of vertices of G and k the number of contractions needed to reduce it to a trivial vertex.*

Proof. The encoding $C(G)$ consists of exactly n 1's, together with elements of a multiset $U \subseteq \{2, 3, \ldots, \Delta^+(G)+4\}$. We know that $C(G)$ starts and ends with an 1, and it contains no two consecutive elements of U. Therefore $C(G) \leq 2n - 1$. When G consists of the induced path P_n, it follows $|C(P_n)| = 2n - 1$, attaining the bound.

Theorem 14 *The isomorphism algorithm terminates within $O(n)$ time.*

Proof. Recall that $m = O(n)$, by Lemma 7. The construction of a bottom-up contractile sequence requires $O(n)$ steps. For each $v \in V(G)$, following the isomorphism algorithm, $C(v)$ can be constructed in time $|C(v)|$. We remark that lexicographic ordering takes linear time on the total length of the strings to be sorted. It follows that the algorithm requires no more than $O(n)$ time to construct the code $C(G)$ of G.

6 Conclusions

The analysis of control flow graphs and different forms of structuring have been considered in various papers. To our knowledge, no full characterization and no recognition algorithm for control flow graphs of structured programs have been described before. There are some related classes for which characterizations and efficient recognition algorithms do exist, e.g. the classes of reducible graphs and D-charts. However, both contain and are much larger than Dijkstra graphs.

An important question solved in this paper is that of recognizing whether two control flow graphs (of structured programs) are syntactically equivalent, i.e., isomorphic. Such question fits in the area of *code similarity analysis*, with applications in clone detection, plagiarism and software forensics.

Since the establishment of structured programming, some new statements have been proposed to add to the original structures which forms the classical structured programming, enlarging the collection of allowed statements. Some of such relevant statements are depicted in Figures 9.

(a) *break-while*: Allows an early exit from a while statement;
(b) *continue-while*: Allows a while statement to proceed, after its original termination;
(c) *break-repeat*: Allows an early exit from a repeat statement;
(d) *continue-repeat*: Allows a repeat statement to proceed, after its original termination;
(e) *divergent-if-then-else*: A selection statement, similar to the standard *if-then-else*, except that the comparisons do not converge afterwords to a same point, but lead to disjoint structures. Note that the corresponding graph has no longer a (unique) sink.

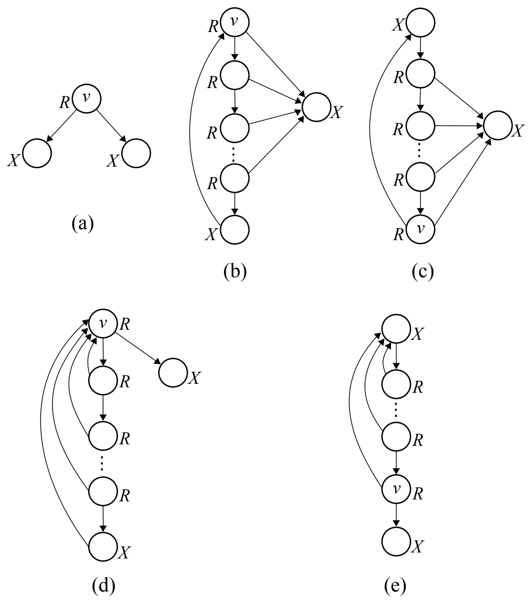


Fig. 9: Generalized Dijkstra graphs

In fact, the inclusion of some of the above additional control blocks in structured programming has been already predicted in some papers, as [17]. The basic ideas and techniques described in the present work can be generalized, so as to efficiently recognize graphs that incorporate the above statements, in addition to those of Dijkstra graphs. Similarly, for the isomorphism algorithm.

Acknowledgments

The authors are grateful to Victor Campos for the helpful discussions and comments during the French-Brazilian Workshop of Graphs and Optimizations, in Redonda, CE, Brazil, 2016. He pointed out the possibility of decreasing the complexity of the recognition algorithm from $O(n^2)$ to $O(n)$.

References

1. F. E. Allen and J. Cocke, A program data flow analysis procedure, *Comm. ACM* 19 (1976), 137–147.
2. A. V. Aho and J. D. Ullman, Node listings for reducible flow graphs, *J. of Computer and Systems Scien.ces* 13 (1976) 286-299
3. L. M. S. Bento, D. R. Boccardo, R. C. S. Machado, V. G. Pereira de Sá and J. L. Szwarcfiter, Towards a provably resilient scheme for graph-based watermarking, *Proc. 39th Intl. Workshop on Graph-Theoretic Concepts in Comp. Sci. (WG'13), LNCS* 8165 (2013), 50–63.
4. C. Böhm and G. Jacopini, Flow diagrams, Turing machines and languages with only two formation rules, *Comm. of the ACM* 9 (1966), 366-371
5. N. Chapin and S. P. Denniston, Characteristics of a structured program, *ACM SIGPLAN Notices* 13 (1978), 36–45.
6. C. Collberg, S. Kobourov, E. Carter and C. Thomborson, Error-correcting graphs for software watermarking, *29th Workshop on Graph-Theoretic Concepts in Computer Science, (WG'03), Lecture Notes in Computer Science* 2880 (2003), 156–167.
7. Collberg, C. and J. Nagra, *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection.* Addison Wesley (2010).
8. E. W. Dijkstra, Go-to statement considered harmful, *Comm. ACM* 11 (1968), 174–186.
9. O.-J. Dahl and C. A. Hoare, Hierarchical program structures, in *Structured Programming*, Academic Press, 1972, 175-220.
10. E. W. Dijkstra, Notes on Structured Programming, in *Structured Programming* (1972), 1–82, Acad. Press.
11. N.E.Fenton, R. W. Whitty and A. A. Kaposi, A generalized mathematical theory of structured programming, *Theoretical Computer Science* 36 (1985), 145-171
12. D. Harel, On folk theorems, *Comm. of the ACM* 23 (1980), 379-389
13. M. S. Hecht and J. D. Ullman, Flow graph reducibility, *SIAM J. on Comp.* 1 (1972), 188-202.
14. M. S. Hecht and J. D. Ullman, Characterizations of reducible flow graphs, *J. ACM* 21 (1974), 367–374.
15. P. Henderson and R. Snow, An experiment in structured programming, *BIT* 12 (1972), 38–53.
16. C. A. R. Hoare, Notes on data structuring, in *Structured Programming*, Academic Press, 1972, 83-174.
17. D. E. Knuth, Structured programming with go-to statements, *ACM Comp. Surveys* 6 (1974) 261–301.
18. D. E. Knuth and R. W. Floyd, Notes on avoinding 'go to' statements, *Inf. Proc. Let.* 1 (1971), 23–31.
19. D. Knuth and J. L. Szwarcfiter, A structured program to generate all topological sort arrangements, *Inf. Proc. Let.* 2 (1974), 153–157.

20. S. R. Kosaroju,, Analysis of structured programs, *J. of Computer and Systems Sciences* 9 (1974) 232-255
21. D. Kozen and W.-L. D. Tseng, The Böhm-Jacopini Theorem is false, propositionally, *Mathematics of Program Construction (MPC' 08)*, CIRM, Marseille, France, *Lecture Notes in Computer Science* 5133 (2008), 177-192
22. T. J. McCabe, A complexity measure, *IEEE Transactions on Software Engineering* SE-2 (1976), 308-320.
23. P. Naur, ed. Report on the algorithmic language ALGOL 60, *Comm. ACM* 3 (1960), 299-314.
24. G. Oulsnam, Unravelling unstructured programs, *The Computer Journal* 25 (1982), 379-387.
25. R. E. Tarjan, Depth-first search and linear graph algorithms, *SIAM J. Comp.* 1 (1972), 146-160.
26. R. E. Tarjan, Finding dominators in directed graphs, *SIAM J. Comp.* 3 (1974), 62–89.
27. R. E. Tarjan, Testing flow graph reducibility, *J. Comp. Sys. Sci.* 9(1974), 355-365
28. N. Wirth, Program development by stepwise refinement, *Comm. ACM* 14 (1971), 221-227.
29. N. Wirth, The programming language PASCAL, *Acta Informatica* 1 (1971), 35-63.
30. N. Wirth, On the composition of well structured programs, *ACM Comp. Surveys* 6 (1974), 247-259.
31. M. H. Williams, Generating flow diagrams:the nature of unstructuredness, *Computer Journal* 20 (1977), 45-50
32. M. H. Williams, Flowchart schemata and the problem of nomenclature, *Computer Journal* 26 (1983), 270-276
33. M. H. Williams and H. L. Ossher, Conversion of unstructured flow diagrams to structured form *Computer Journal* 21 (1978), 101-107.
34. W. A. Wulf, A case against the Go-to, *Proc. of the ACM Ann. Conf.*, 1972, 791-791.